LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The VAL Intermediate Graph Format

Computation Structures Group Memo 235
12 January 1984

William B. Ackerman

This research was supported by the Department of Energy under grant number DE-AC02-79ER10473 and by the National Science Foundation under grant number MCS-7915255.

Source File = CSG230.MSS.6, Last updated 12 January 1984 at 3:16pm

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

1. INTRODUCTION

An intermediate graph-like format has been designed for internal use by VAL compilation systems. The VAL compiler in use at MIT uses this internal format. The full compiler is actually several programs -- a parser, linker, code generator, and cell allocator. The linker writes its output in essentially this format, which the generator then reads. The format has been carefully (after some trial and error) chosen to be the correct format for data flow program transformations and optimizations. That is, program structures are represented in a manner amenable to the transformations of interest. Furthermore, the format is used at a point in the compilation process far enough along that aspects of the program that are not relevant to the transformations have been removed and other irrelevant aspect (things that depend on the target machine) have not yet been put in. Some simple optimizations have been performed on programs represented in this format by an experimental optimizer, and it is expected that much more sophisticated optimizers will use it in the future.

In this paper, the format will be described in an informal graphical way, using pictures. The concrete format used in the files produced by the VAL linker and optimizer is dependent on the CLU implementation of those programs and will be described in another memo.

The format is a graph corresponding to the structure of the data flow program. Each node corresponds to one operator (arithmetic, array or record operation, etc.) or one complex structure (conditional, **tagcase**, **for** loop, or **forall**).

2. LINK STRUCTURE

Links exist between nodes when there is a data path from one node to the other. In the internal computer format, these links are bidirectional, so each node possesses a pointer to the other.[1] The

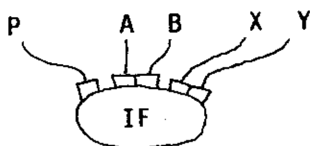
---

1. In the actual file written out by the VAL linker, the back links are not yet in. They are added by the optimizer after reading in the file.

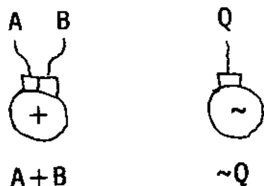
information about each link contained in the two nodes is enough that, starting from just one of the nodes, the link can be completely removed or modified. Each node has some input links and some output links associated with it. The input links point to the other nodes that send data to this node, and the output links point to the other nodes to which data are sent.

The inputs of a node are organized into *arms*, with each arm divided into *args*. For a simple **if-then-else** of arity two, there are three arms, with 1, 2, and 2 args respectively.[1] This will be diagrammed as shown:

```
if P then A, B else X, Y endif
```



Ordinary arithmetic operators have only one arm, with some number of args in it.



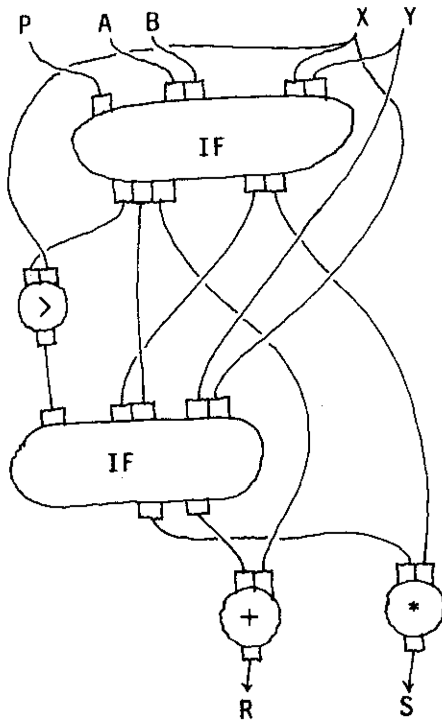
The outputs of a node are organized into *results*, with each result divided into *arcs*. For a structure of high arity, there are as many results as the arity. For each result, there are as many arcs as there are places to which that result is to be sent. Ordinary operators have only one result, with some number of arcs.

---

1. This is actually a lie. The true nature of the **if** block will be clarified later.

A typical interconnection might look like this

```
T, U := if P then A, B else X, Y endif ;
W, Z := if T > X then U, T else Y, X endif ;
R := Z+T ;
S := W*U ;
```



Each group of boxes along the top of a node corresponds to an arm, and each box is an arg. Each group of boxes along the bottom corresponds to a result, with each box corresponding to an arc.

In the actual data structure for a node, each incoming link (box along the top) has three pieces of information with it:

1. The node pointed to
2. Its result number

3. The arc number for that result

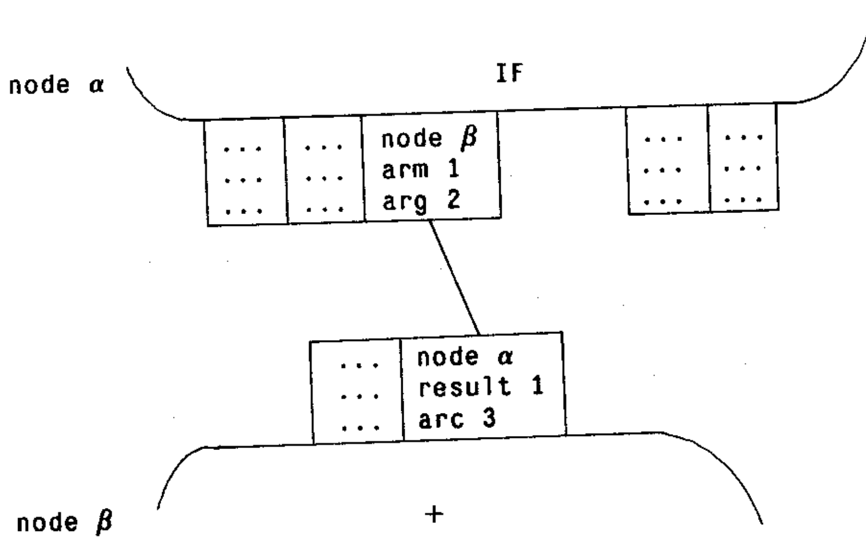
Note that this is sufficient information to locate the pointer at the other end of the link, if it is desired to modify the link.

Each outgoing link also has three pieces of information:

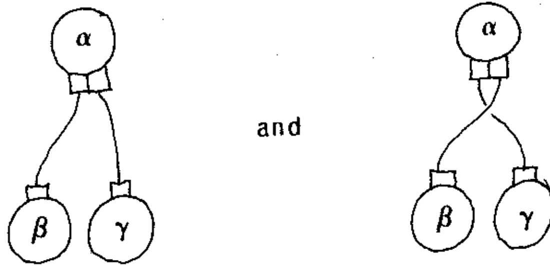
1. The node pointed to
2. Its arm number
3. The arg number in that arm

Note that this is also sufficient information to locate the pointer going the other way.

A typical link, viewed under a microscope, might look like



The assignment of arcs for a result is of course arbitrary -- the same result goes to all destinations. The following are equivalent:



The back pointers must be consistent in all cases. The assignment of args for an arm is of course not arbitrary.

The input data associated with each node comprise an array of arrays of triples. These triples are records containing

node - pointer to a node

result - integer

arc - integer

(the type and symbol name are here also and will be discussed below)

This array of arrays can not necessarily be implemented as a two-dimensional array, since the arms do not all have the same number of args, and hence the lower level arrays are not always the same length. The CLU implementation of arrays works well here.

The output data associated with each node comprise an array of arrays (of different lengths -- the preceeding paragraph applies here also) of triples. The triples are

node - pointer to a node

arm - integer

arg - integer

3. TYPE INFORMATION

Another piece of information associated with each input link is the input data type. This is a pointer to the global canonicalized type list. No information on result types is stored. It can be deduced from the operator, and could, in any case, be found by tracing the output link and examining the input link at the other end.

4. DIAGNOSTIC INFORMATION - SYMBOLS

There is diagnostic information, that is not essential to compilation or execution. but is an aid to debugging and analysis. This information consists of symbol names and line numbers. The intention is that, if an overflow occurs in

```
X := P*Q ;
```

and the user has requested a diagnostic printout if this happens, something will be printed such as

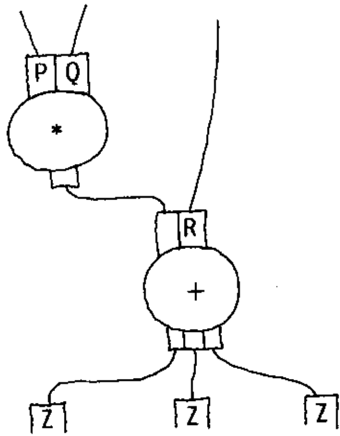
```
*** pos_over[real] created by multiply ***
        1st arg (P) = 3.14e29
        2nd arg (Q) = 2.72e21
at line 4 of function foo
called from line 13 of function bar
called from line 27 of function main
```

The symbol information is stored with the input links. In addition to the node, result, and arc, there is a 4th item -- the name. This is an index into the global symbol table, which is merged from all of the symbol tables of all of the modules.

The name may be absent. If we have

```
let
     P := ... ;
     Q := ... ;
     R := ... ;
     Z := P*Q+R ;
```

the graph is



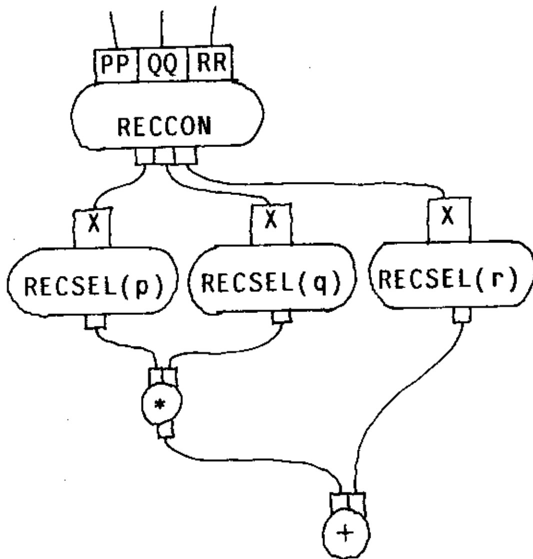
The left argument of the addition has no symbol.

A common operation in VAL will be the packing of many values into a record, binding that record to a name (perhaps sending it to a function) and them selecting and using the fields. If we have

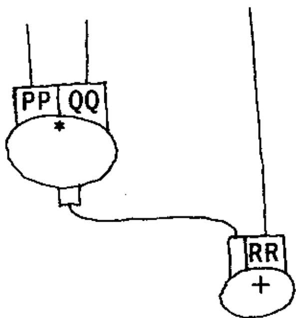
```
PP := ... ;
QQ := ... ;
RR := ... ;
Z := foo(record[p: PP; q: QQ; r: RR]) ;
.
.
.

function foo(X: ...)
     X.p * X.q + X.r
```

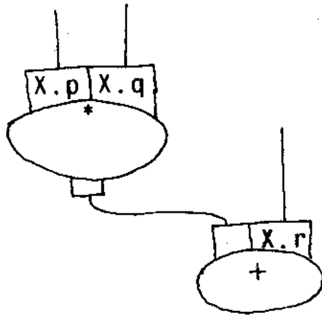
then, assuming static function expansion, the graph might look like



Now a common optimization would be the removal of record construct/select pairs. If this is done preserving the symbol names, we would get



This is not a good idea, because a diagnostic printout would show symbols that are actually defined in another function. We want the printout to show what the arguments looked like at the place in the text where the operator appeared. Therefore, the symbol information stored with an input link can have record field selectors, and the correct optimization of the above graph should yield



A similar sort of thing can happen with array selects using a constant index, though this will be far less common. So the symbol information can have arbitrary record fields and array indices, and a diagnostic printout of an argument might look like

```
X.p[7].q[6][10]
```

This is stored as an array of symbol table indices and array index values. This array is the name part of each input link. An empty array signifies a nameless argument.

5. DIAGNOSTIC INFORMATION - LOCATION AND CONTEXT

The rest of the diagnostic information is the location of the operator and its context. This information is associated with each entire node, not with the individual links. It consists of the line number and the function, or the line number and the file. (The two implementations are equivalent -- given the file and line number, one can tell what function it is in.)

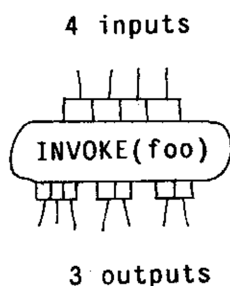
When functions are dynamic, that is, not expanded statically by the compiler at every invocation, this is all the location information that is present in the node. To print out the context information (where the function was called from, all the way back to the top-level function), some run-time data will be needed, which must be obtained by decoding the "stack", or the equivalent structure in a data flow computer.

If functions are statically expanded, the nodes in each expansion must give the line number and the context, instead of line number and function. The context is a pointer into the *context table.* Every time a function is expanded by a graph transformation program, an entry is created in the context table giving the called function, line number (in the calling function) from which it was called, and a pointer to the entry in

the context table for the calling function. When a run-time error occurs, it will be possible to print the full context by chasing pointers in the context table.

6. FUNCTION INVOCATION AND EXPANSION

The output of the VAL linker does not expand functions, so that it can be used with dynamic data flow projects. This output is a collection of graphs, one for each function. Invocations of functions appear as node types, giving the function:



An optimizer or code generator for a static machine must remove all these nodes, replacing them with copies of the indicated subgraph. The present experimental VAL code generator does this.

Inputs to functions are represented by an *arg* node type, which has only outgoing links. Outputs are represented by a *result* node type, which has only incoming links. The table of functions has pointers to these special nodes for each function -- they are the "handles" for the functions.

7. NODE TYPES

There are the following general node types. In addition to the information specific to each node type, all nodes have the line numbers and context information described above, and the input and output links.

Elementary operators -- these are the "scalar" operators of the VAL language

Special operators -- these are operators that the compiler makes available as though they were library functions because the hardware is capable of executing them directly. Examples of such operations are logical and arithmetic shifts, square root, and trig functions.

Compound operators -- these are the array, record, and oneof operations. It is these nodes that an optimizing compiler is (or should be) interested in.

Function invocations

Arg and result nodes

Constant values

if, **tagcase**, **iterif**, **itertag**, **for**, and **forall** blocks -- these are described below.

Note that **let** blocks are not represented here. They are expanded as part of the process of putting the program into this graphical form. A block such as

```
let P := X-Y in P*P endlet
```

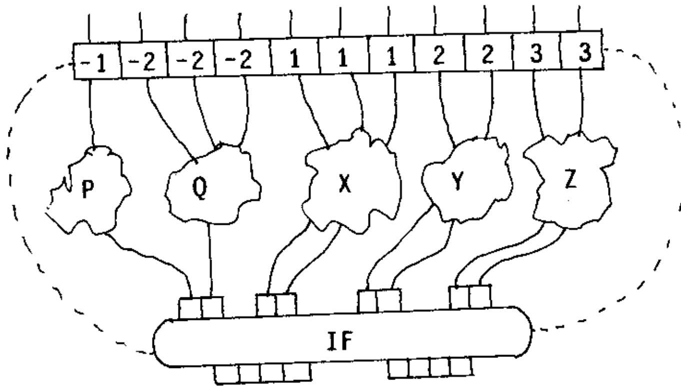
is represented as

8. COMPLEX STRUCTURES - IF

Complex structures -- **if**, **tagcase**, **for**, **forall**, and the structures for iteration bodies, have subparts that have tokens flowing through them perhaps many times, and perhaps not at all, for a given operation. It is necessary to generate appropriate token control instructions (gates, merges) at the boundaries of the subparts. For example, the tests and result clauses of an **if** block need gates to allow tokens into only the selected clause.

An **if** is represented this way, where P, Q, X, Y, and Z are arbitrary subgraphs:

```
if P then X
elseif Q then Y
else Z
endif
```



The boxes at the top are called *gates*, and the numbers inside them are the *case* numbers. All inputs to the subgraph for the ith clause of the **if** pass through a gate with case `i`. All inputs to the subgraph for the predicate enabling the ith clause pass through a gate with case `-i`.

All of the control values enter the **if** in one arm. Since control values have arity 1, it would be wasteful to give them separate arms. That arm is arm number 2, for reasons that will be discussed below. The clauses of the **if** each use an arm of the node, starting at arm 3. Those arms have as many inputs as the arity of the **if**.

Nothing can get to a subgraph without going through an appropriately numbered gate. All subgraphs are completely inside the structure for the **if**. There are no "leaks" across the dotted line, nor cross-links among different subgraphs.

The gates are actually part of the **if** node. They are drawn in a row at the top, connected to the actual node by dotted lines, only to make the graph geometrically reasonable. In the computer representation, they are part of the node, and their inputs and outputs are inputs and outputs of the node. The reason for doing this is so that all data links related to the **if** will have a uniform structure.

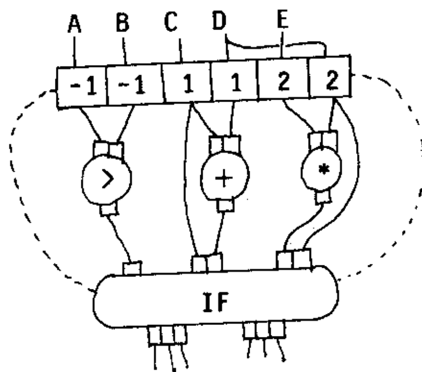
The gates are handled as follows: Their inputs collectively comprise arm number 1. This is why the control values went to arm 2 and the clauses went to arms starting at 3. The number of inputs for arm 1 is just the number of gates. The outputs of the gates are all distinct outputs of the node. The leftmost gate's output is output 0. It has as many arcs as needed to convey its data to of all the places that require it. The next gate is output number -1, and so on, using negative numbers.

Positive outputs are thus true outputs of the **if-then-else** structure. Nonpositive outputs go to internal substructures. Arm 1 receives true inputs to the **if-then-else** structure. Higher numbered arms come from internal substructures. There are no nonpositive arm numbers. The cases are kept in a separate array associated with the node.

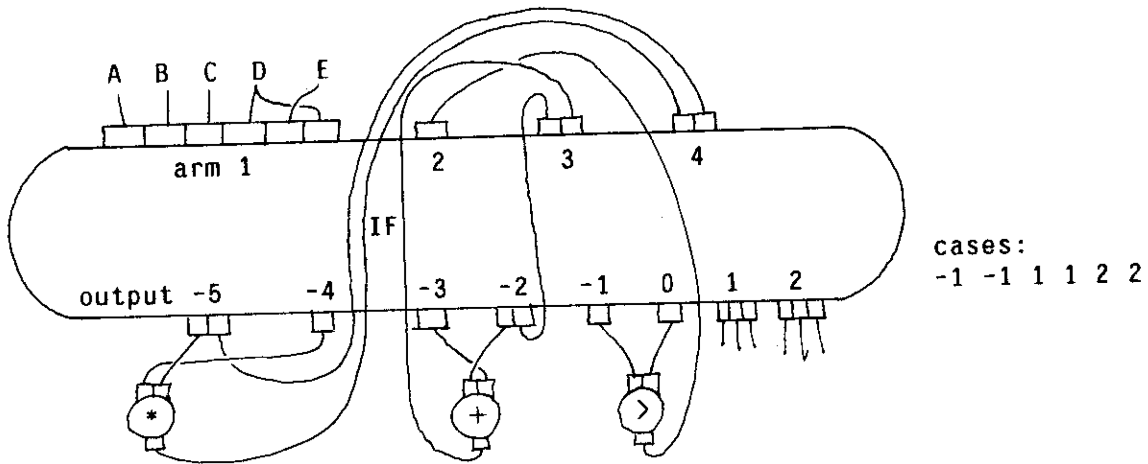
A reasonable diagram for

```
if A > B then C, C+D else E*D, D endif
```

would be



A fanatically faithful diagram would be



9. TAGCASE

This is similar to **if** except as follows:

1. Arm 2 (the one bearing the boolean test values in an **if**) has only one arg. That arg is the *oneof* datum to be tested. This datum does *not* pass through any gate. It goes from the external source of the datum directly into arg 1 of arm 2.

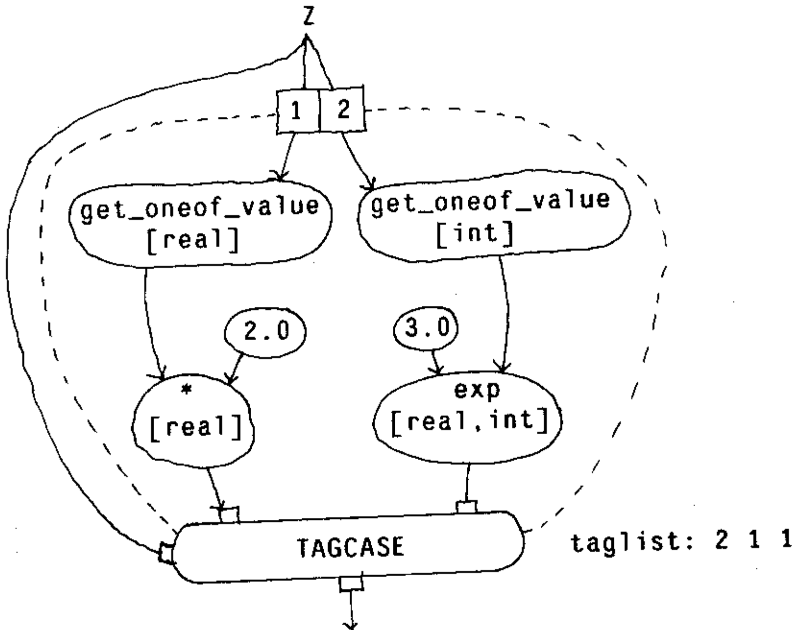
2. The **tagcase** node contains another piece of data -- the *taglist*. This is an array of integers. If the *oneof* datum has tag `i` (all oneof and record types have canonical numbers assigned to their tags and fields, which numbers are used in the type table and in all operations) then `taglist[i]` gives the clause to be evaluated.

3. The **tagcase** block makes the constituent value of the *oneof* type available inside the arms. The operation of extracting the constituent value is a separate "`get_oneof_value`" operator in the graph. This operator yields a result whose type is fixed. It is the type of the constituent value of the oneof datum. There is a field in the operator node giving that type. The compiler guarantees that, since this operator will only appear within certain arms of a tagcase, the tag of the incoming datum will always be such that the constituent value will be of the indicated type.

Example:

```
type T=oneof[q,r:real; i:integer];

tagcase P := Z                         % Z is of type T
tag q,r: P*2.0
tag i: exp(3.0,P)
endtag
```



Since the canonical order for the tags of type T is (i, q, r) the taglist denotes the mapping i→2, q→1, r→1.

10. ITERIF AND ITERTAG

These are nodes that may evaluate to an **iter** clause in a loop. They are almost identical to **if** or **tagcase**, except that their clauses are classified into three categories: "yes", "no", and "maybe".

"yes" clauses always iterate. Their arity is the number of iteration variables in the loop, and the values going into a "yes" arm are the new values to be assigned to the loop variables, in the order in which those variables are listed in the **for** block. A clause labeled "yes" does not have to be an **iter** block in the source program -- it could be a complex conditional all of whose clauses are **iter** blocks. That is, the VAL linker effectively turns

```
if P the iter I, J := F, G enditer
else iter I, J := S, T enditer
endif
```

into

```
iter I, J := if P then F, G else S, T endif enditer
```

All of the loop variables need not be assigned in an **iter** block. The semantics of VAL states that unassigned variables retain the same value on the next cycle. The VAL linker takes care of this. All "yes" clauses provide the new values of *all* loop variables, just giving the old value where necessary. The linker also takes care of the distinction between "new" and "old" values of loop variables in an **iter** block. When a loop variable appears on the right hand side of an assignment in an **iter** block, it refers to the "new" value if that variable has been reassigned in an assignment appearing earlier in the **iter** block; otherwise the "old" value is used. The VAL compiler and linker know about this and create the correct data flow graph going into the "yes" arm of the **iterif** or **itertag**. If the VAL language is changed to use some other mechanism to identify "new" and "old" values, the intermediate graph representation will not have to change.

"no" clauses never iterate. They indicate that the loop is to exit. Their arity is the arity of the **for** loop, and the values of the clause are the values to be returned from the loop.

"maybe" clauses may or may not iterate. They are special -- their arity is always one, even though many values may be returned or reassigned. The data arc going into a "maybe" arm comes only from another **iterif** or **itertag**.

The classification of clauses as "yes", "no", or "maybe" is stored in the *controls* array of the node.

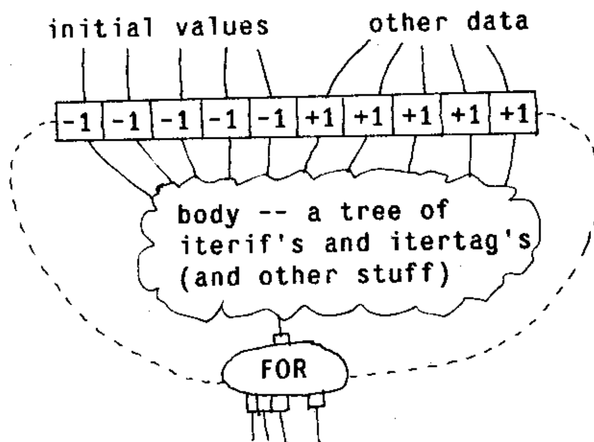
All **iterif** and **itertag** nodes always appear to return one value. (That is, their only positive output is number 1; they still have nonpositive outputs from their gates.) The single output value has only one arc, and it must go to a "maybe" arm of another **iterif** or **itertag** node, or to the single arm of a **for** node.

11. FOR

This node has the body of the **for** loop as arm 2, and the gate inputs as arm 1, as for the other nodes with gates. Arm 2 is, except in pathological cases,[1] a "maybe" arm linked from an **iterif** or **itertag** node. The **for** has a *controls* array with one entry which is "maybe" except in pathological cases.

This node has a number **N** of loop variables. (The number **N** is one of the data fields of the node.) The leftmost **N** gates are for these loop variables. The inputs to these gates have the *initial* values of the variables, and the gates have case `-1`. All reassignments in an **iter** clause ("yes" arm anywhere inside the body) contain the new values in the same order. The remaining gates have case `+1` and are for "free variables" of the loop, that is, data passing into the loop that are not loop variables and do not change value from one cycle to the next.

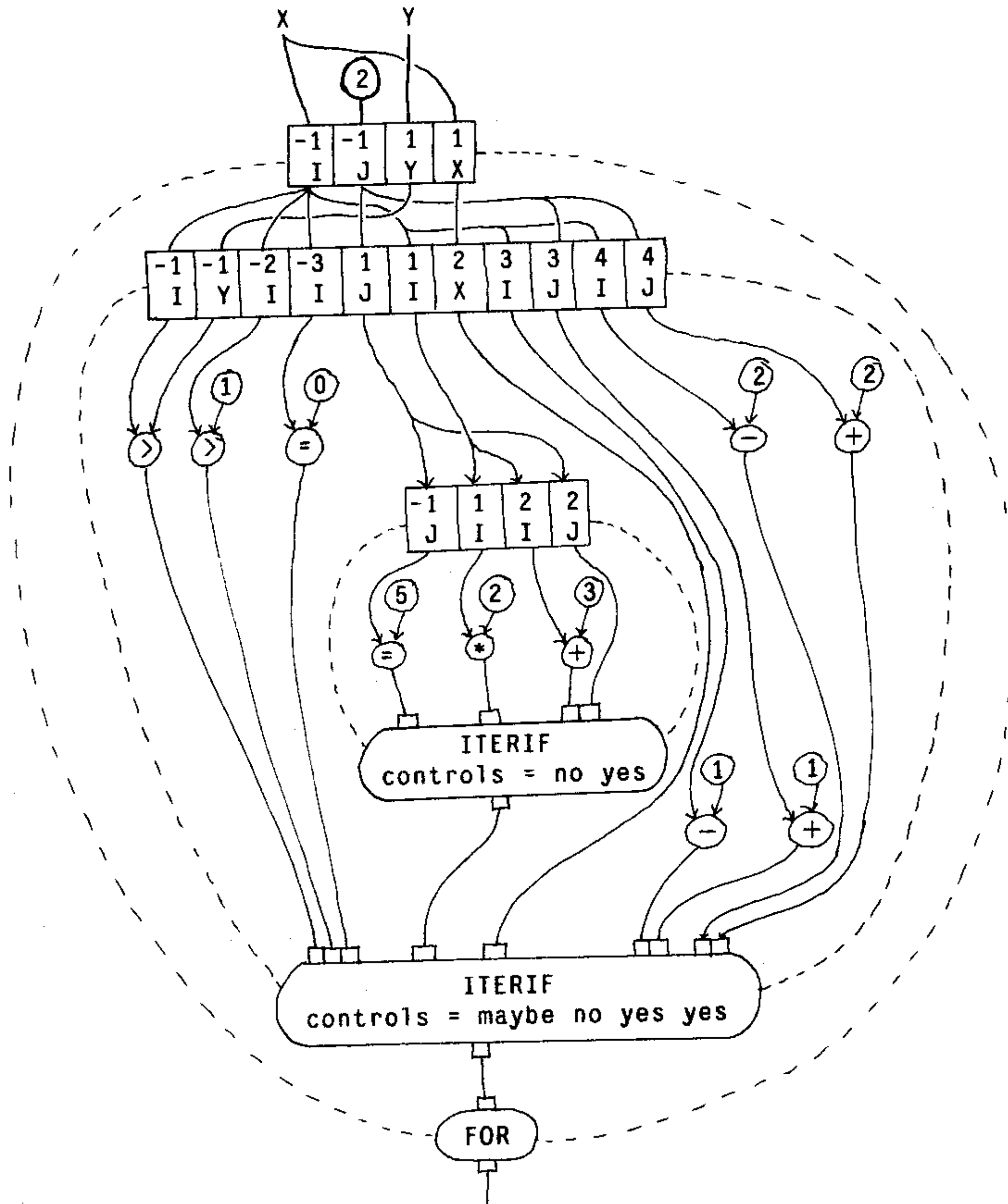
There are as many positive-numbered outputs of the **for** node as there are values returned from the loop. As usual, nonpositive outputs are outputs from gates.



1. "`for I := 0 do 7 endfor`" is such a case. It body is classified as "no". A "yes" body such as "`for I := 0 do iter I := I+1 enditer endfor`" is illegal.

Example:

```
for I, J := X, 2
do   if I>Y then
          if J=5 then I*2 else iter I := I+3 enditer endif
     elseif I>1 then X
     elseif I=0 then iter I, J := I-1, J+1 enditer
     else iter I, J := I-2, J+2 enditer
     endif
endfor
```



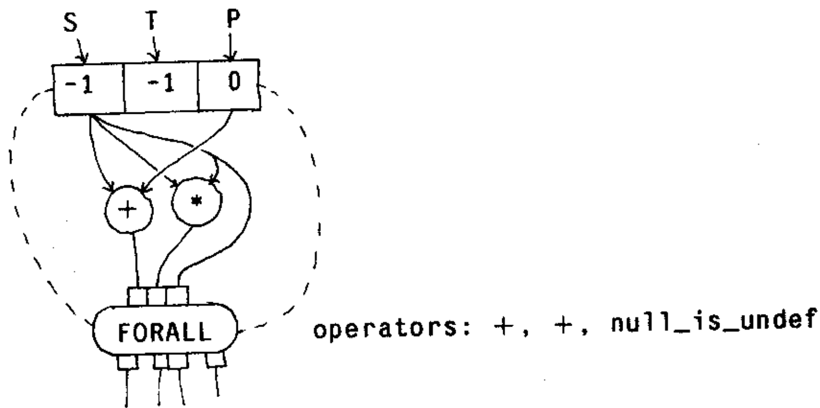
12. FORALL

This node has the body as arm 2, and the gate inputs as arm 1, as for the other nodes with gates. The first two gates receive the low and high index limits. These gates have case **-1**. The output of the first gate is the index variable, and goes to wherever it is needed in the body. The second gate has no output arcs.

For each value of the index variable between the limits, the body must be evaluated, yielding a number of values equal to the arity. Those values are to be combined according to a series of operators -- one operator per arg. The list of operators is part of the data of the **forall** node. The operators are the ones that are legal in **eval** clauses, plus the operator "null_is_undef" for **construct** clauses.

Data other than the index variable, that is,

```
forall I in [S, T]
eval plus I+P, I*I
construct I
endall
```



Multi-index forall's, as in

```
forall I in [P, Q], J in [R, S]
. . . . .
```

are decomposed into separate nested **forall's**. The **forall** node in the graph always denotes a one-dimensional **forall**.